The Legion Support for Advanced Parameter-Space Studies on a Grid*

Anand Natrajan, Marty A. Humphrey, Andrew S. Grimshaw
Department of Computer Science, University of Virginia
{anand, humphrey, grimshaw}@cs.virginia.edu

Abstract. *Parameter-space studies involve running a single application several times with different parameter sets. Since the jobs are mutually independent, many computing resources can be recruited to conduct an entire study in a distributed manner. Parameter-space studies are attractive applications for grids, which are networked collections of computing and other resources. Legion is a grid infrastructure that facilitates the secure and easy use of heterogeneous, geographically-distributed resources by providing the illusion of a single virtual machine from those resources. Legion provides tools and services that support advanced parameter-space studies, i.e., studies that make complex demands such as transparent access to distributed files, fault-tolerance and security. We demonstrate these benefits with a protein-folding experiment in which a molecular simulation package was run over a grid managed by Legion.*

1. Introduction

A *computational grid* or a *grid* is a collection of distributed heterogeneous resources connected by a network. Grids are becoming more pervasive platforms for running distributed jobs to solve large problems. In such an environment, users can access resources transparently and securely. When a user submits jobs in a grid, the system runs them on distributed resources and enables her to access her results during execution and on completion. In a grid, users are not limited by geography, by non-possession of accounts, by limits of resources at one site or another and so on. In short, as long as a resource provider is willing to permit a user to use the resource, there is no barrier between the user and the resource.

Increasingly, grids are becoming platforms for running large parameter-space (p-space) studies. In the past, grids have been viewed as platforms for high-performance applications with multiple communicating tasks distributed across several machines. However, current network latencies, although greatly reduced from what they were in the past, still do not support running communication-intensive tasks across different machines; the latencies in communication reduce performance significantly. Therefore, the new focus of grid infrastructures is to support high-performance computing by running large p-space studies. Since in p-space studies, individual tasks do not communicate with one another, high performance can be achieved by ignoring network latencies largely although not entirely.

This paper describes the support for p-space studies provided by Legion, a grid infrastructure. Legion is a software infrastructure for managing a grid [10]. Legion provides an abstraction of the grid similar to what a traditional operating system provides for a single machine. This abstraction supports the current performance demands of scientific applications. Legion supports capacity computing, i.e., the ability to conduct larger computational experiments either by expending more resources on a single problem or on multiple, independent problems, as well as capability computing, i.e., new mechanisms with which to conduct computational science experiments. A number of scientific applications already run using Legion as the underlying infrastructure [22]. In the future, scientists will demand support for new methods of collaboration. Legion supports these expected demands as well [21]. The aim of this paper is to show that Legion provides most of the current requirements of p-space studies: data management, binary management, scheduling, fault tolerance, transparent remote execution of legacy and non-legacy applications, distributed file systems, security and a job monitoring interface.

In §2, we briefly present and discuss some of the underlying philosophy of Legion. In §3, we present Legion's support for p-space studies. In this section we first discuss Legion's support for running single applications; running p-space studies can be viewed as running single applications several times over. In §4, we present a case study that demonstrates Legion's support for running high-performance parallel legacy p-space applications on cross-organisational, heterogeneous, distributed and potentially-faulty resources. The results of this experiment clearly indicate that an integrated grid infrastructure is essential for enabling users to take advantage of grid resources. In §5, we present the current

* This work was supported in part by the National Science Foundation grant EIA-9974968, DoD/Logicon contract 979103 (DAHC94-96-C-0008) and by the NASA Information Power Grid program.

Report Documentation Page

Form Approved
OMB No. 0704-0188

1. REPORT DATE	2. REPORT TYPE	3. DATES COVERED
2006		**00-00-2006 to 00-00-2006**

4. TITLE AND SUBTITLE	5a. CONTRACT NUMBER
The Legion Support for Advanced Parameter-Space Studies on a Grid	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)	5d. PROJECT NUMBER
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **University of Virginia,Department of Computer Science,151 Engineer's Way,Cahrlottesville,VA,22094-4740**	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSOR/MONITOR'S ACRONYM(S)
	11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

13. SUPPLEMENTARY NOTES
The original document contains color images.

14. ABSTRACT

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			
unclassified	**unclassified**	**unclassified**		**16**	

status of Legion's support for p-space study. The support is influenced by our experiences with users who run high-performance p-space studies. We conclude in §6.

2. Legion Background

The Legion project is an architecture for designing and building system services that present users the illusion of a single virtual machine [10]. This virtual machine provides secure shared objects and shared name spaces. Whereas a conventional operating system provides an abstraction of a single computer, Legion aggregates a large number of diverse computers running different operating systems into a single abstraction. As part of this abstraction, Legion provides mechanisms to couple diverse applications and diverse resources, vastly simplifying the task of writing applications in heterogeneous distributed systems.

Legion is a grid operating system. It provides standard operating system services — process creation and control, interprocess communication, file system, security and resource management — on a grid. In other words, Legion abstracts the distributed, heterogeneous and potentially faulty resources of a grid by presenting users with the illusion of a single virtual machine [12]. In order to achieve this goal, Legion manages complexity in a number of dimensions. For example, it masks the complexity involved in running on machines with different operating systems and architectures, managed by different software systems, owned by different organisations and located at multiple sites. In addition, Legion provides a user with high-level services in the form of tools for specifying what an application requires and accessing available resources.

Each system and application component in Legion is an object. The object-based architecture enables modularity, data and fault encapsulation, and replaceability — the ability to change implementations of any component. Legion provides persistent storage, process management, resource management services, security and inter-process communication, long regarded as the basic services any operating system must provide. Legion provides these services in an integrated environment, not as disjoint mechanisms such as Globus does [7]. Of particular importance is the integration of security into Legion from the design through implementation. Legion supports PVM [8], MPI [27], C, Fortran (with an object-based parallel dialect), a parallel C++ [9], Java and the CORBA IDL [26]. Also, Legion addresses critical issues such as flexibility and extensibility, site autonomy, binary management and limited forms of fault detection/recovery. From inception, Legion was designed to manage millions of hosts and billions of objects — a capability lacking in other object-based distributed systems [13].

A well-designed grid should not only satisfy current demands of users but also anticipate and satisfy future demands. Currently, many applications require high performance. However, in the near future, grid systems such as Legion will be able to deliver high performance to applications routinely by providing access to distributed resources. We believe that at that point, users will look beyond high performance as the defining feature of a grid system. At that point, users' demands may include heterogeneity, security, fault-tolerance and collaboration.

Heterogeneity is a fundamental design principle in Legion [11]. Typically, a running grid that uses Legion incorporates diverse resources — machines of different architectures running different operating systems consisting of different configurations and managed by different organisations. Legion users can register implementations of different architectures for their applications. For parallel applications, different tasks started by a single job may run on heterogeneous machines and communicate with one another as if they ran on homogeneous machines. For p-space studies, different jobs of an application may be started on machines of different architectures.

Security was designed into Legion from the start [6]. Every Legion object, whether it be a resource, a user, a file, an application or a running job, has a security mechanism associated with it. The mechanisms provided by Legion are general enough to accommodate different kinds of security policies within a single grid. Typically, the security provided is in the form of access control lists. An access control list indicates which objects can call which methods of an object. This fine-grained control mechanism enables users and grid administrators to set sophisticated policies for different objects. The authentication mechanism currently employed by Legion is a public key infrastructure based on key pairs. The keys are used to encrypt and decrypt messages securely as well as for signing certificates.

Fault-tolerance can be implemented in a number of ways in Legion [24]. Basic Legion objects are fault-tolerant because they can be deactivated at any time. When a Legion object is deactivated, it saves its state to persistent storage and frees memory and process state. Subsequently, it may be reactivated from its persistent state either on the same or a different machine. If it is reactivated on a different machine, Legion transfers its state to the new machine whenever possible. In addition, some objects can be replicated for performance or availability. Legion's MPI implementation provides mechanisms for checkpointing, stopping and restarting individual instances. Finally, Legion provides tools for retrieving intermediate files generated by legacy applications. Users can restart their instances using these

intermediate files. For p-space studies, individual jobs can be monitored and restarted anew if they fail.

Legion enables new paradigms for collaboration between researchers conducting experiments that require using grid resources. We believe that collaboration is an important goal for a grid system. We believe that researchers should not be limited by geographical distance between one another as well as the resources they desire to use. Accordingly, the ability to share objects via their permissions (access control lists) has always been a key design feature in Legion.

3. Support for Parameter-Space Studies

Legion can benefit scientific applications by delivering large amounts of resources such as computing power, storage space and memory. Moreover, Legion provides a rich set of tools that make the access and use of these resources simple and straightforward. In particular, there are tools for running programs written using MPI or PVM as well as programs that are p-space studies or sequential codes. In §3.1, we show how Legion enables a user to access large amounts of computational resources (which are usually controlled by queuing systems at various organisations) for running high-performance applications. In §3.2, we discuss information services and scheduling in Legion briefly. In §3.3, we present Legion's distributed file system, called context space, which can be used to access data distributed across several machines in a transparent manner. In §3.4, we present some of Legion's tools for running applications, especially tools for running p-space studies of applications. In §3.5, we discuss advanced features of runs, such as sharing jobs among mutually-distrustful users.

3.1. Hosts, Vaults and Queuing Systems

Resources such as machines, disks and queues are represented by corresponding objects in Legion. The representations of these resources abstract the particular details of using them. Consequently, users are presented with a uniform interface for utilising these resources. The resource objects also encapsulate local policies about their usage. For example, a host object may encapsulate policies about which users can run on the underlying machine. Likewise, a vault object may encapsulate policies about how much disk resources can be made available to a user. A rich set of resources can be abstracted by Legion objects; in the rest of this subsection, we will discuss the benefits of abstracting one particular kind of resource, namely, queuing systems, in Legion.

Queuing systems have been used to schedule jobs on many clusters of nodes [2] [17] [19] [30] [31]. When a user submits a job, the queue provides a ticket or job ID or token, which can be used to monitor the job at any later time. The ticket becomes invalid shortly after the job completes. Most queuing systems comply with a POSIX interface requiring three standard tools for running jobs: a submit tool (PBS `qsub`, LSF `bsub`, LoadLeveler `llsubmit`), a status tool (PBS `qstat`, LSF `bjobs`, LoadLeveler `llstatus`) and a cancel tool (PBS `qdel`, LSF `bkill`, LoadLeveler `llcancel`). In addition, some queues provide other tools to check on the aggregate status of the queuing system, e.g., LSF `bqueues` and LoadLeveler `llq`. A queuing system's status tool may report that a job is queued, running or terminated. If the execution of a job is deemed undesirable, the cancel tool can be used to terminate the job. Most queuing systems do not provide tools to access intermediate files or supply additional inputs. A user desiring such functionality must employ shared file systems or other file transfer tools. Queuing systems do not provide any support for checking aggregate progress of large sets of jobs. Users must check on the progress of each job individually or construct interfaces to monitor the progress of the entire set of jobs.

Part of the abstraction Legion provides is to hide the differences among queuing systems as well as between queuing and non-queuing systems. A user running over Legion does not have to know the particulars of every system on which a job could run. To appreciate why this abstraction is important, consider running a simple application, such as "Hello, world", on different systems. We could run on a Unix or Windows system by writing a shell script or batch file such as the one in Figure 1.

```
echo 'Hello, world'
```

Figure 1. Simple application

However, if we wanted to run on a cluster of nodes controlled by Portable Batch System (PBS), we would have to modify the application to construct a submission script as in Figure 2. If we decided to run on nodes controlled by Maui/LoadLeveler, we would have to construct a submission script as in Figure 3. Not only are

```
#!/bin/ksh
#PBS -A anand
#PBS -c n
#PBS -m n
#PBS -N LegionObject
#PBS -r n
#PBS -l nodes=1:ppn=1:walltime=00:10:00
#PBS -p 1
#PBS -o test.o
#PBS -e test.e

echo 'Hello, world'
```

Figure 2. Simple application modified for PBS

different queuing systems dissimilar, but the same queuing system installed at different sites may be dissimilar in

```
#!/bin/ksh
# @ environment = COPY_ALL;MP_EUILIB=us
# @ account_no = met200
# @ class = express
# @ node = 1,1
# @ tasks_per_node = 1
# @ wall_clock_limit = 00:10:00
# @ input = /dev/null
# @ output = test.o
# @ error = test.e
# @ initialdir = /home/uxlegion

echo 'Hello, world'
```

Figure 3. Simple application modified for Maui

terms of configuration parameters. Moreover, the tools for running special applications, e.g., MPI programs, may be different (`mpirun` *versus* `pam` *versus* `poe`). The different submission scripts required to run on different systems restrict a user in two significant ways:

1. The user is forced to learn the particulars of each queuing system, thus increasing his cognitive burden and increasing the time before he can start becoming productive on these systems.
2. When running large numbers of jobs, the user must construct submission scripts for running on each queuing system. The very act of creating a submission script *a priori* forces the user to construct a static schedule for running his jobs. Consequently, he cannot take advantage of dynamic load changes on resources to schedule jobs.

Legion hides differences among queuing systems regarding their submit, status and cancel tools as well as their submission scripts. Also, Legion hides differences regarding the manner in which MPI jobs are run. Moreover, Legion provides tools and mechanisms for accessing intermediate files and viewing the aggregate status of large numbers of jobs. Finally, Legion does not require the user to log on to the various queuing systems to initiate jobs. Single sign-on is one of the most convenient features of a grid operating system.

3.2. Information Services and Scheduling

Scheduling is the process of running jobs on the best possible resources on a grid. The general scheduling problem is NP-complete [28]. In addition, the parameters involved in making an optimal schedule are numerous and mutually dependent. Constructing a schedule may involve making decisions not limited to: (a) the machine architectures for which a class has implementations, (b) specific properties of a machine desired by the class (e.g., is it a queuing system? can it run MPI jobs natively?), (c) communication bandwidth *versus* performance penalty, (d) current load and storage space on the machine, (e) permissions for this user to run an instance of this class on that machine, (f) allocation remaining for the user on that machine and (g) charges imposed by resource providers for running on their machine.

Legion provides mechanisms to construct schedulers. Different schedulers may employ different algorithms to construct schedules from the list of available resources. Also, Legion permits users to specify resources directly for a job, the rationale being that until good heuristics are developed to address all issues in scheduling, users are likely to be the best schedulers of their own jobs. For p-space studies, Legion provides a tool called `legion_make_schedule` that constructs schedules based on requirements for the application as well as performance characteristics of the available (and working) machines on which the user is permitted to run.

The general scheduling architecture in Legion is based on negotiation between resource providers and consumers [4]. The negotiation process preserves autonomy of resource providers while satisfying the demands of the consumers. When a user starts a job, Legion encapsulates the demands of the user in the job request. The scheduler uses this request to construct one or more schedules for this job. Next, it queries the resource objects in turn to determine if they will accept the job. The resource objects may exercise the autonomy of the resource providers in accepting or denying the job. If they accept, the jobs are initiated on the chosen resources.

Scheduling is an example of a situation requiring information services. Such services collect and store information about interesting components of a grid. In Legion, an information service is represented as an object called a collection. The collection collects and stores information about other objects; the choice of objects is a configuration issue. For example, in a particular grid, a collection may be configured to collect information about users (represented as user objects), such as email address, postal address, last login, preferences, etc. Another collection may be configured to collect information about computing resources, such as machines. This collection would either pull information from the host objects or have information pushed into it from the host objects. The information could be static, such as the operating system and architecture of the host, as well as dynamic, such as load, available memory, available swap space, etc. The information in such a collection could be (and is) used by a scheduler object to assign jobs on machines.

3.3. Distributed File System: Context Space

Legion provides a shared, virtual space to grid users. The shared, virtual space can be viewed as a truly distributed, global file system. Components of this file systems are visible to all Legion users from any of the machines that are part of the grid. For example, if a job that is part of a p-space application is scheduled on one of

the machines that is part of the grid, the machine can access the files necessary for that job although the files may be stored on some other machine. These accesses occur without the user's intervention.

The distributed file system is organised in a manner similar to a Unix file system. In order to distinguish the global file system from the file systems present on individual machines, we call the global file system a *context space*. Directories in context space are called *contexts*. A context called "/" typically denotes the root of the context space. A context is an object that contains other objects — contexts, hosts, schedulers, users, classes, files, etc. All users of a grid, no matter where located physically, have the same view of the context space. The analogue of this model in traditional operating systems is an NFS-mounted disk that is visible to all machines that share the mount, or a Samba-mounted Unix directory that is visible from a Windows machine.

The scope of Legion's context space is much vaster than that of any of its predecessors. Distributed file systems are not novel. Legion's implementation has predecessors in Network File System (NFS) [25], the Andrew File System (AFS) [16] and Extensible File System (ELFS) [18]. However, context space is truly distributed and global; individual components may be physically located on machines that do not have anything in common except that they are part of the same grid.

Users may freely transfer files from their local file systems to context space. For example, one of the options to a tool called `legion_cp` permits users to copy a text file from their file system to context space. Likewise, registering a program effectively transfers an executable from a local file system to context space. A growing number of tools available in Legion permit users to interface with context space in novel ways. For example, a tool called `legion_export_dir` lets a user mirror an entire directory in his local file system into Legion. This tool is particularly useful for enabling large parameter sets to be accessible to several jobs running on different machines using Legion. Likewise, a Windows tool lets users browse context space. When these two tools are used in conjunction, a user on one Windows machine may be able to view the contents of his collaborator's directories on another Windows machine across the globe. Naturally, the permissions on the exported directory and its components have to be set to permit the collaborator (and perhaps only the collaborator) to view them. However, setting the permissions is a matter of manipulating the access control lists of the objects. Legion provides tools for manipulating the access control lists of objects.

Tools for traversing context space include a suite of Unix-like command-line tools, a point-and-click Web browser interface, an FTP tool, a Samba interface for Windows, an HTTP interface, and a Legion implementation of NFS for accessing context space with standard Unix tools such as `ls` and `cat` as well as with standard system calls like `open`, `read` and `write` [29]. Using these tools, grid users can collaborate by sharing and exchanging data in a manner familiar to them. Moreover, because of the possibility of setting fine-grained access controls, collaborators can also select the level of collaboration.

3.4. Running Single Applications

In Legion, running an application typically requires two steps. The first step, called "registering", requires the user to supply Legion with the binaries for the application. The second step, involves the actual execution of the binaries on different resources. Registering binaries is a necessary one-time step whereby the user lets Legion transfer the correct binaries to the machines (and only those machines) on which the user eventually runs. Binary management is a useful feature for users who desire to access large numbers of resources for their application. Given that users often may not even be aware of the machines on which their applications run, it becomes the responsibility of the grid infrastructure to transfer the appropriate binaries on exactly those machines on which the application runs. In the following subsections, we discuss Legion's support for various kinds of applications.

Legacy Applications. Legion supports running legacy applications on a grid. Legacy applications are those whose source code does not consist of any calls to Legion routines and does not utilise Legion objects and tools. Moreover, the source code of the application may not be modified to target it to Legion, either because it is unavailable or because its authors are unavailable or unwilling to make the necessary changes. In all such cases, Legion neither mandates re-targetting the application nor denies access to grid resources.

Legacy applications are not targetted specifically to a grid or Legion. Legion supports such applications "as is", i.e., the user neither has to change a single line of code nor re-link the object code to run such an application. All Legion requires are the executables for the application for various architectures. A user who chooses this form of support understands the trade-offs for the convenience of not changing the application at all. One trade-off is that Legion can control very few aspects of the execution of the job after it is initiated. For example, Legion cannot provide restart support for a legacy application if the application itself does not write checkpointing data. However, Legion can and does provide support for starting the job, checking its status as reported by the underlying system and terminating the job if necessary. In addition,

Legion provides the ability to send in or get out intermediate files while the job is running.

A Legion user may run a legacy application on the distributed resources of a grid by undertaking two steps (tool names are in parentheses): register the executable as a runnable class (`legion_register_program`) and run the class (`legion_run`). The first step results in the creation of a *runnable class*, analogous to an executable in Unix or Windows. Registering an executable is an infrequent step, required only when the runnable class does not exist in Legion or when the executable available to the user changes. A user is likely to execute the second step repeatedly in order to initiate, monitor and complete repeated runs of the application. The executable registered with this class is called an *implementation*. Multiple executables, typically of different architectures, may be registered with the same class.

Once a runnable class has been created in Legion, a user can run the class by issuing a `legion_run` command. The simplest form of the command is:

```
legion_run myClass
```

Here, the user implies that Legion can run an instance of the class on any resource present in Legion provided (a) `myClass` has implementations for the machine on which the instance eventually runs (e.g., Solaris or SGI implementations), (b) the user is permitted to run on the machine, and (c) the machine accepts the instance for running. More sophisticated runs can involve the user specifying machines or architecture types on which she would like to run, input and output files as well as meta-information regarding how a job should be run on a particular machine. Legion ensures that the input and output files are copied to and from the machine on which the instance runs. Moreover, if the user desires, she can observe the on-screen output of the remotely-executing job on her current terminal. In keeping with the Legion philosophy of providing mechanisms on which policies can be constructed, there exist many different strategies for executing a legacy application on distributed resources. These different strategies can be applied by choosing from a large number of options available in `legion_run`. The options are part of the standard documentation and man pages available at each Legion installation [1].

MPI Applications. Many high-performance parallel applications are written using the Message Passing Interface (MPI) library [27]. An MPI library provides routines that enable communication among various processes of a parallel application. MPI is a standard, i.e., it defines the interface of the routines. Different vendors of MPI may implement a routine differently provided they adhere to the standard interface. Legion's support for MPI is three-fold: Legion MPI, native MPI and mixed MPI.

Legion MPI. Legion can be viewed as another MPI vendor because it provides implementations to standard MPI routines. If a user desires to run an application that uses MPI routines on a grid, he has to undertake three simple steps: re-link the object code of the application with Legion libraries (`legion_link`), register the executable as an MPI runnable class (`legion_mpi_register`), run the class (`legion_mpi_run`). The first step ensures that Legion's implementation of MPI routines are used when running the application. Note that it is not necessary to change the source code of the application. The subsequent steps are similar to those for legacy applications. The options and operations of the actual commands are similar to those for registering and running legacy applications.

Native MPI. Some MPI applications are intolerant of high latencies for inter-process communications. Running such applications on distributed resources may degrade the performance of the application. Such applications are better supported by running them on proximal resources to reduce communications latency. Moreover, many MPI implementations are tuned finely to exploit the architecture of underlying resources. Finally, the users of many MPI applications may be unwilling or unable to re-link the application with Legion libraries. Therefore, Legion supports running MPI applications in "native" mode, i.e., using other implementations of MPI, such as MPICH [14]. Native MPI support is similar to support for Legion MPI as well as legacy applications. The steps a user has to undertake are: register the executable as a runnable class (`legion_native_mpi_register`) and run the class (`legion_native_mpi_run`). The benefits to the user are that no recompiling or re-linking is necessary to access remote resources transparently. We describe an example of Legion's support for native MPI applications in §4.

Mixed MPI. In Legion's mixed MPI support, an application is executed in "native" mode, but the application can access Legion's objects, such as files. The steps required are: modify source code to initiallise Legion library, re-link the object code with Legion libraries (`legion_link`), register the executable as a runnable class (`legion_native_mpi_register`) and run the class (`legion_native_mpi_run -legion`). The user has to modify the source code to initiallise Legion with one call from within the application. Registering and running the class is similar to native MPI with the addition of one option. Applications written to take advantage of mixed MPI support can benefit in two ways: (a) since jobs are executed in native mode, performance for latency-intolerant applications does not suffer, and (b) jobs can access Legion objects and thus take advantage of the grid.

Mentat and Basic Fortran Support (BFS). High-performance applications can be supported in Legion if they are written in Mentat or if they use the Basic Fortran Support. Mentat is a language similar to C++ with a few additional keywords [9]. In Mentat, users may specify classes to be stateless or persistent. The Mentat compiler identifies data dependencies within a program and constructs a dataflow graph to execute the program. Mentat provides a platform for users to write high-performance applications using a compiler constructed to mask the tedium of writing parallel programs. Legion's support for Fortran programs is called BFS [5]. If users desire to write grid applications in Fortran, then Legion requires that grid directives be embedded within Fortran comments. Currently, BFS support targets Mentat, but may not in future releases.

Parameter-Space Studies. Many grid applications are p-space studies. In a p-space study, a single program is called repeatedly with different sets of parameters. Multiple instances of the program may run concurrently with different sets of parameters. These instances are completely independent of one another. Therefore, they can be scheduled easily across geographically-distributed resources. With Legion's support, users may run their p-space studies orders of magnitude faster than sequential. First, the application must be registered. Next, the user must indicate which files must be mapped to the files required by an instance. Finally, the application must be run with `legion_run_multi`. Legion runs each instance of the application by mapping the proper files for the instance and copying output files appropriately. `legion_run_multi` takes a number of options in order to tailor the running of a p-space application for a user. This tool ensures that input files and output files are arranged such that the user can identify corresponding sets easily. Legion provides tools for viewing the progress and performance of a p-space study (`legion_show_multi` and `legion_plot_multi` respectively).

Legion's support for p-space studies involves determining the requirements of the application, picking the best machines to run on and controlling the flow of jobs such that high throughput is achieved. In addition, `legion_run_multi` checks whether the appropriate output files for each job have been retrieved and restarts jobs that complete unsuccessfully. In addition, if the user so desires, he can view the progress of each and every job. In Figure 4, we show part of the view a user sees when running p-space studies. On the left is a view showing 1000 jobs of an application with 250 jobs running at any time. Since this view of a realistic p-space study is somewhat dense for explanation, on the right we show a view of 75 jobs of a similar application with 25 running at any time. Each horizontal line shows the progress of a job along the time axis (left is earlier in time). The colour of a line encodes the state of that job, e.g., blue indicates that the job is actually running on a remote machine, green indicates that the job is done, etc. The labels on the left of each line denote the ID of that job. A user can observe the progress of any job from the corresponding line. The vertical lines at the bottom show how many jobs were running at any given time. Ideally, the number of jobs running at all times should be equal to the number specified by the user (25 in this case). However, when the p-space application is started and when it is completing, the number of jobs running changes. Likewise, at times, the tool does not manage to keep the maximum number of jobs running because it has to cycle through the completed jobs to retrieve output files and start new jobs in their place. In Figure 5, we show a snapshot of the full screen visible to a user when the application is in progress. On the right is the view similar to the one shown on the right in Figure 4. The buttons at the bottom left can be used to obtain more detail about any particular job. For example, in the snapshot, the user has pressed the button corresponding to the job labelled "41". The text window at the middle left shows details about the job, e.g., its status and the machine on which it is running. The buttons at the top left can be used to obtain details about any particular machine, e.g., which jobs are currently in progress on it. The machines selected for this particular job are selected based on load from the entire set of machines available on this grid (*npacinet*).

Currently, a limitation with Legion's support for p-space studies is that the user must specify the number of jobs that can be in progress concurrently. The user does not have to specify the total number of jobs; that number is deduced by `legion_run_multi` as the number of complete sets of input files less the number of complete sets of output files. However, Legion does not deduce the number of jobs that must be concurrent. The number of concurrent jobs cannot be made arbitrarily high for several reasons: (a) too many jobs can swamp the client from which the p-space jobs is initiated in terms of CPU usage, process table usage, memory, etc. (b) if the duration of a single job is shorter than the time it takes to start several jobs, then starting more jobs concurrently does not result in better throughput (c) from an accounting viewpoint, the number of concurrent jobs may be throttled because the user cannot afford to pay for using so many resources concurrently (d) initiating too many jobs may overload available machines. Deducing the number of jobs that must be run concurrently is an interesting area of future research in running p-space applications on a grid.

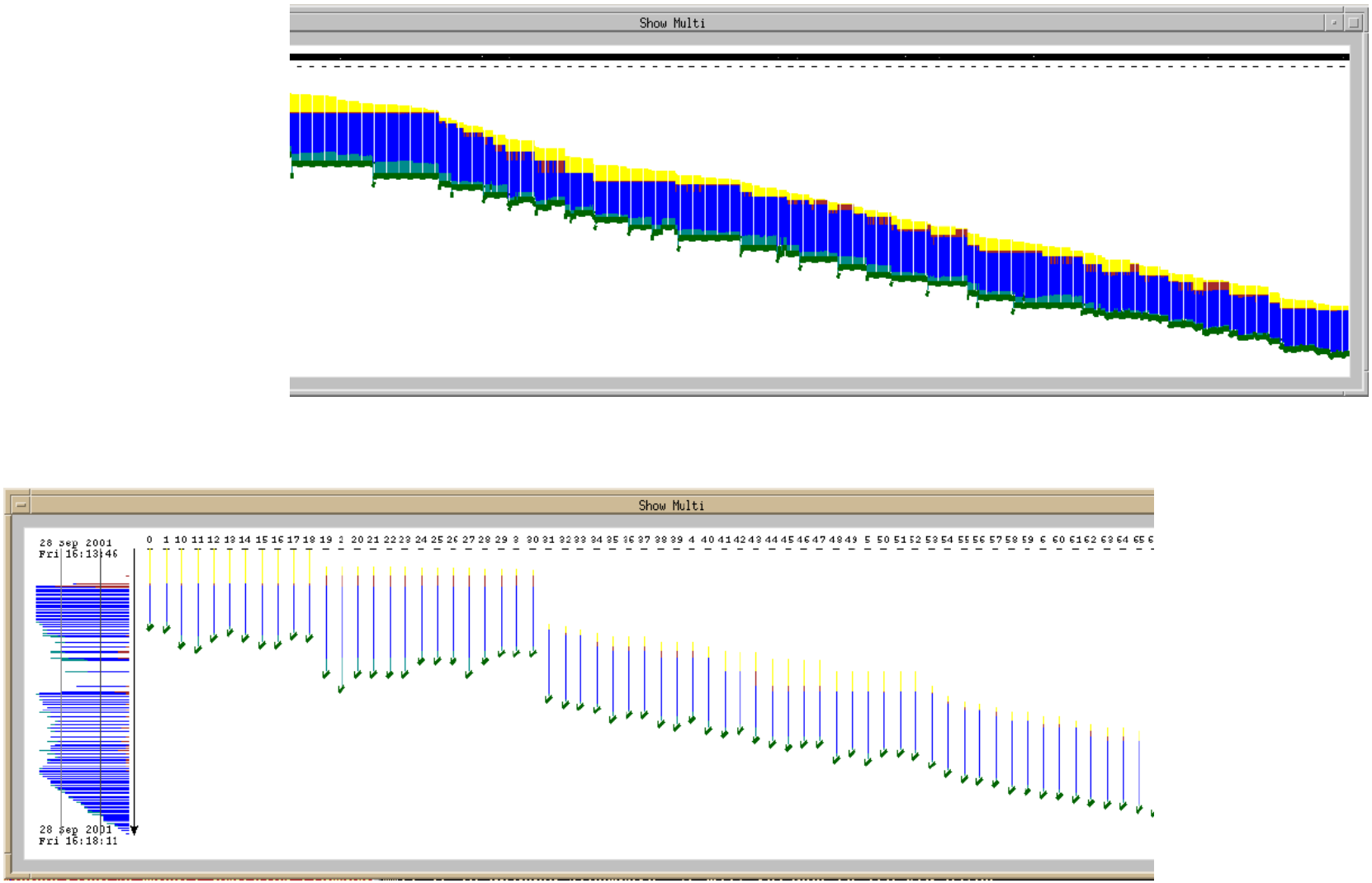

Figure 4. Progress of Parameter-Space Studies

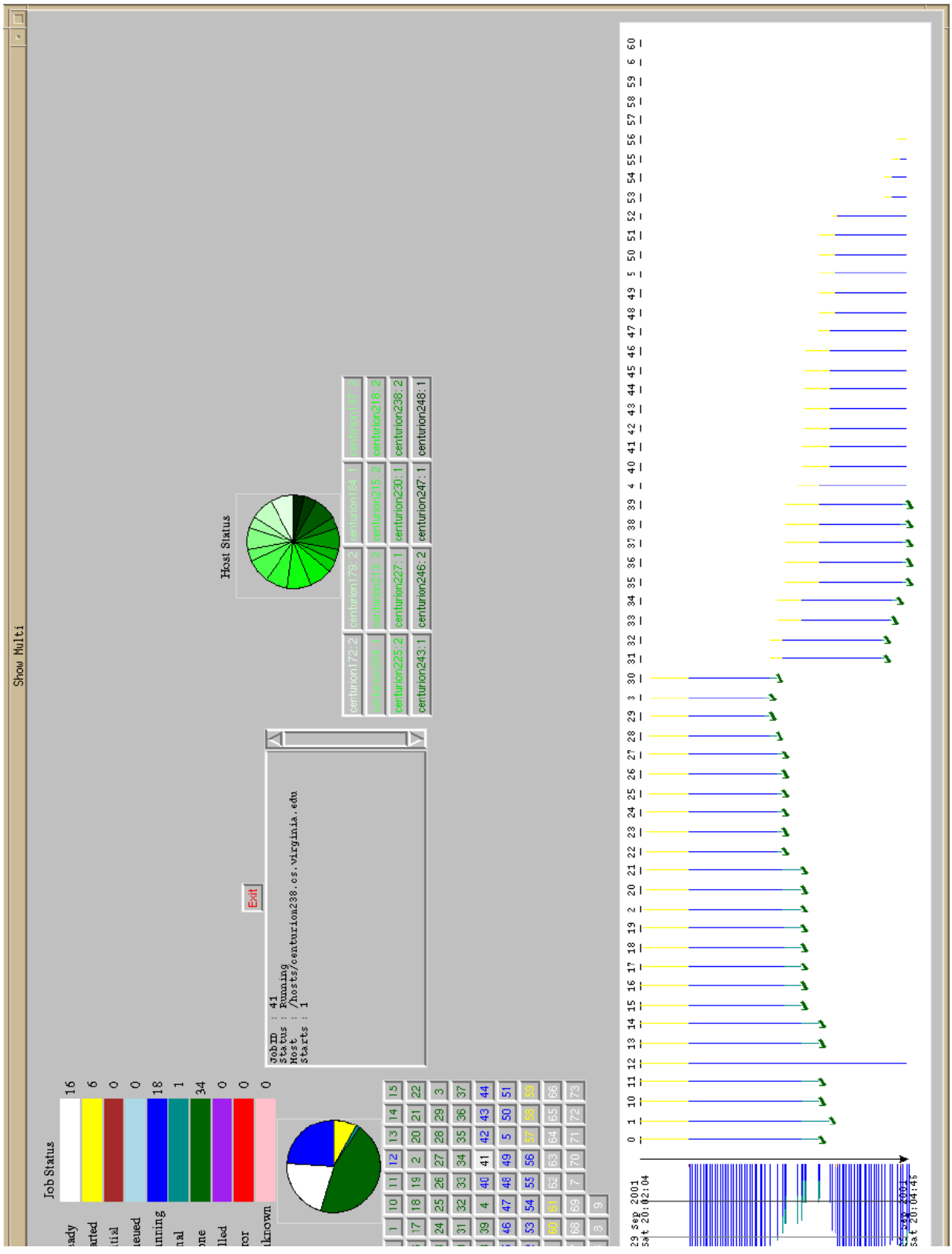

Figure 5. Full Display of Small Parameter-Space Study

3.5. Sharing Jobs

Legion's object model is flexible enough to permit novel means of collaboration among researchers, for example, sharing jobs. In Legion, running instances of a class are first-class objects themselves. Therefore, as with any object in Legion, access control lists can be set for them to control permissions in interesting ways.

Suppose two researchers situated across a country wish to collaborate. The nature of their collaboration requires one of them to initiate a job which both observe. Currently, such a collaboration would be impossible unless both researchers were able to share an account on some machine. In Legion, neither researcher would need an account on the machine on which the instance runs. Instead, both could access the same object using Legion tools from their own machines.

Suppose a researcher constructs an application that is used widely by others in the same field. The researcher could register her executable as a runnable class in Legion and set the permissions to allow anyone, a group of users or an *a priori* known set of users to run instances of the class. Currently, the researcher would have to send or sell her executable to her fellow researchers. In the Legion model, she could control who runs her class when, where and how many times without physically transporting her executable to the other researchers' machines.

Suppose two mutually-distrustful parties wish to collaborate on an experiment with one providing the executable and the other the data. Currently, such a collaboration is impossible because either the executable or the data must be transported to the other collaborator. However, in Legion, such a collaboration is legitimate and possible. The collaborator with the executable would register the executable as a class in Legion and start an instance. Then he would set the permissions on the instance allowing only the other collaborator to perform data transfers but retaining permission to terminate the experiment. The second collaborator, after verifying that the permissions are indeed as outlined above, could commence transferring data files. The application in question would have to be written in such a manner that it can wait until the data files become present. With that minor change in place, Legion can enable these mutually-distrustful parties to collaborate.

Other means of collaboration will become evident as grids are used more widely and routinely. We expect the Legion model to be flexible enough to accommodate these collaboration efforts as they arise.

4. Case Study: Running CHARMM on NPACI Resources using Legion

In order to demonstrate Legion's support for high-performance p-space computing, we conducted an experiment in which a computational scientist accessed resources from NSF's National Partnership for Advanced Computational Infrastructure (NPACI) using the grid infrastructure provided by Legion. The application used was CHARMM (Chemistry at HARvard Molecular Mechanics) [3] [20], a popular general simulation package used by molecular biologists to study protein and nucleic acid structure and function. One large problem for which CHARMM is used is the study of the nature of the protein folding process. The scientist desired to study the energy and entropy of many folded and unfolded states of a certain protein, Protein L, to gather information about its behaviour during its folding process and to generate a protein-folding landscape. This study required multiple CHARMM jobs to be run with different initial parameters.

There were two clear goals for this experiment:

1. *Enhance the productivity of the user by solving a large and computationally-challenging problem.* By accessing distributed grid resources, the user condensed the time required for performing his computations from a month (if he used the resources available at his organisation) to less than two days.

2. *Demonstrate a match between mechanisms expected by the user and those provided by the grid infrastructure.* The user had to learn five commands or fewer in order to perform his computations on a variety of resources.

In the process of meeting these goals, we made a number of observations that affect grid infrastructure developers as well as grid users. In this paper, we present those observations in the context of the experiment.

Our primary observation was that grid infrastructures must provide high-level services in addition to low-level functionality. Providing low-level functionality alone is not enough; without high-level services built on top of the underlying infrastructure, a user's productivity can fall tremendously. The novelty of this experiment is not the solving of a large problem, but the ease with which the user accessed grid resources and the low cognitive burden imposed on him by the grid infrastructure, Legion. In this paper, we describe how the user interacted with a grid using Legion services, what problems arose with the resources that were part of the grid and how Legion addressed those problems, and what lessons we learned regarding new functionality that can be provided to users.

4.1. CHARMM

The protein folding process is not well-understood and the state-of-the-art methods of studying it are too computationally intensive to be undertaken often. One method is to calculate the free energy surface of the folding process. The calculation is designed to reveal the process by which a small protein (Protein L) folds up into its normal, three-dimensional configuration. The folding process occurs in nature every time a protein molecule is manufactured within a cell. The biophysics of folding must be understood in detail before the information can be used in developing ways of interacting with proteins to cure diseases such as Alzheimer's or cystic fibrosis.

The CHARMM molecular simulation package uses the CHARMM force field to model the energetics, forces and dynamics of biological molecules using the classical method of integrating Newton's equations of motion. Typical systems studied involve protein or nucleic acid molecules of several hundred to several thousand atoms and a bath of solvent, usually water, consisting of many thousands of molecules, for a total of 20000 to 150000 atoms. All chemical bonds and all interactions that do not involve bonds (for example, electrostatics) are used to model the system. These interactions number in the millions to billions. For a typical simulation, hundreds of thousands to millions of timesteps of integration are required and, at each timestep, all interactions are determined. In a parallel job, all forces and all coordinates must be shared among all processors.

CHARMM is computation- as well as communication-intensive. In a single CHARMM job, hundreds of processes may perform computations and communicate with one other. The processes communicate using Message Passing Interface (MPI), a standard for writing parallel programs [15] [27]. The parallel efficiency of the computation depends on the number and speed of the processors, and the speed and latency of the interconnect. Since processor speed has increased but interconnect speed has lagged on current-generation high-performance computers, CHARMM's performance degrades rapidly after 32 processors on almost all architectures except the T3E, on which it scales well to 128 processors. Therefore, we chose to run with 16 processors on most architectures, getting better than 95% parallel efficiency throughout the experiment on all high-performance architectures. For a 16-processor job, all processors communicate about 1 Mbyte of data at every timestep in a couple of all-to-all communications, and another 4 Mbytes in each-to-each communications. For a typical 16-processor job on a 375MHz Power3, approximately 3 timesteps occur per second. Each job requires a number of input files, some of which are a few Mbytes large, and generates a number of output files, some of which are hundreds of Mbytes large. Thus, a single job requires powerful computation resources, fast network capabilities and large amounts of disk space. In our experiment, the user required multiple (up to 400) CHARMM jobs to be run.

We decided to run the CHARMM jobs on a computational grid because the total amount of computing resources required made it unattractive to run at a single site. Typically though not necessarily, supercomputing centres such as the San Diego Supercomputing Center (SDSC) use queuing systems to control powerful computation resources connected by fast networks. Since such resources are exactly what CHARMM jobs require, our experiment was conducted on queuing systems. Nothing in CHARMM requires a queuing system; our choice of resources was governed by the coincidence that the kinds of resources that CHARMM requires are usually controlled by queues.

4.2. NPACInet

The grid chosen for running CHARMM was *npacinet*, a nation-wide grid consisting of heterogeneous resources present at multiple sites and administered by different organisations. The majority of the organisations contributing resources to *npacinet* are part of NSF's National Partnership for Advanced Computing Infrastructure (NPACI) thrust. Legion has been managing this grid continuously for several months during which we have demonstrated Legion features numerous times, conducted tutorials on multiple occasions and supported various academic users running a variety of applications.

4.3. Steps for Running CHARMM on NPACI Resources

The steps the user had to undertake to run CHARMM over Legion are illustrated in Figure 6. All of these steps were performed after the user logged on (in the Unix sense) to a machine on which Legion had been installed. The shaded boxes represent the steps the user performed without Legion's help. Of these, two, "Creating Jobs" and "Analysing Results", are specific to the application. The third, "Creating Executables", could have been performed with Legion's help. The user had to learn one new Legion command for each of the unshaded boxes. Learning four commands is a small price to pay for the ability to run multiple parallel jobs on distributed heterogeneous resources in a secure and fault-tolerant manner.

Creating Executables. In this step, the user created the executables for CHARMM. Recall that the user chose Legion's legacy MPI support for CHARMM. If he desired, he could have used `legion_make`, a tool to compile the source code on machines or architectures of his choosing (in which case, he would have done so after "Logging on to Grid"). The resulting executables would still be legacy

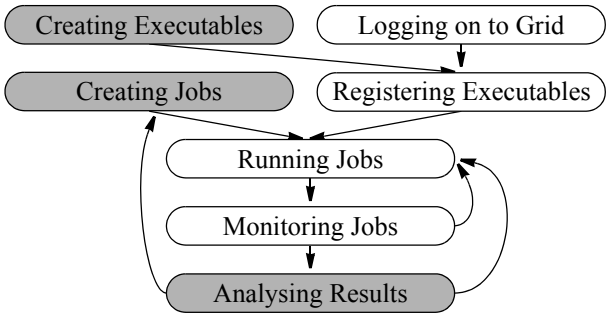

Figure 6. Steps for CHARMM over Legion

code because Legion would not require changing the source code or linking the object code against Legion libraries. Currently, `legion_make` works for applications with relatively simple and standard make rules, i.e., it works for applications that use standard compilers and have straightforward local dependencies. Since CHARMM is not such an application, the user decided to compile for different architectures without Legion's help.

Creating Jobs. This step involved creating a set of input files for each job. Clearly, this step is application-specific and requires no help from Legion.

Logging on to Grid. In order to log on to the *npacinet* grid, the user ran the command `legion_login`, which required him to enter his Legion ID and password. Once the user logged in, Legion did not require him to log on to any other machine.

Registering Executables. Registering executables is the process by which Legion can run a Unix or Windows executable. After an executable is registered with `legion_register_program`, Legion has the information necessary for selecting the appropriate executable to run on any particular machine. Multiple executables of different architectures may be registered with the same Legion object. The benefit is that a user can request Legion to run the object without having to manage which executable should copied and run on which machine. For example, Legion will ensure that only a Solaris executable is copied and run on a Solaris machine.

Running Jobs. After registering the executables for every architecture of interest, the user requested Legion to run the object with the command `legion_run`. This command has a number of parameters and options (details are in the Legion man pages accompanying the standard distribution [1]). Parameters for this command include the name of the object and parameters for the job, the names of input and output files for the job, and options such as number of nodes desired, tasks per node desired, duration, etc. Reasonable defaults are chosen for unspecified options. The user may specify a particular machine on which to run or let Legion choose the machine. Likewise, the user may choose to run on any machine of a particular architecture or let Legion make that decision.

The CHARMM user specified the input and output files for each job and the machines on which he desired to run. In addition, he specified the name of a "probe file" for monitoring the job. The user ran the `legion_run` command as many times as he wanted to initiate jobs. Although he chose different machines on which to run different jobs (effectively self-scheduling his application dynamically), at no point did he have to write a single submit script, log on to any other machine[*], copy executables and input/output files, or learn a new command for running jobs. The user could have initiated as many jobs as he desired concurrently; in practice, he initiated a few tens of jobs concurrently because i) the nature of the jobs imposed sequential dependencies, and ii) initiating multiple jobs is pointless when the next job is certain to be queued behind previous ones.

This particular user did not use our tool for p-space studies, `legion_run_multi` because he wished to have greater control over each run. At the time of the experiment, `legion_run_multi` did not offer the kind of control that this user desired.

Monitoring Jobs. The user monitored each job in two ways. First, he started a console object for the Unix shell from which he initiated his jobs with one command, `legion_tty`. After the console object was started, output and error messages printed by the user's jobs or the queuing systems on remote machines became visible on the user's shell. Second, the user requested Legion to save a probe for every job. Using the probe and a tool called `legion_probe_run`, the user determined the status of each and every job as well as sent in and got out intermediate files at his leisure. If at any time the user determined that a job was not progressing satisfactorily, he terminated it, corrected any problems and restarted it.

Analysing Results. The final step involved analysing the results from each job. A basic analysis step involved determining whether each job actually ran to completion. The user made this determination by checking whether a certain output file contained specific lines in it. A large part of the subsequent analysis involved retrieving archived files and processing them by running CHARMM again. The subsequent steps were specific to the application and are outside the scope of this discussion.

[*] In fact, in the current configuration of Legion on the NPACI machines, the user was not even required to own accounts on the machines. Legion ran his jobs as a generic user on those machines. In the future, the NPACI resources may insist that the user can run on their machines only if he has an account on them as well. Since respecting site autonomy is a critical part of the Legion philosophy, support for the latter mode of operation is under progress.

4.4. Results from Case Study

The experiment was conducted successfully over a period of two days. The user logged in to one machine at the University of Virginia on which Legion was installed*. From a single shell on that machine, he initiated as many jobs as he could, subject to the limitations discussed earlier. Some of the jobs failed, but a large number ran to completion successfully. Consequently, although the user did not manage to complete all of the jobs he desired initially, a significant fraction of the jobs were completed. The experiment showed the viability of running large, high-performance applications on a computational grid. In the following sections, we discuss how well Legion met the goals mentioned earlier.

Increasing User Productivity. A success of this experiment was that the grid was used to generate results for an actual scientific study. At the time of writing, around 88 of the desired 400 jobs had been completed. We demonstrated that Legion can be used to harness a vast amount of processing power harnessed for scientific users. In the final tally, 1020 processors of different architectures and speeds were utilised for this experiment. The breakdown of these processors is:

- 512 375MHz IBM Blue Horizon Power3s at San Diego Supercomputing Center (SDSC)
- 128 440MHz HP PA-8500 at California Institute of Technology (CalTech)
- 24 375MHz IBM SP3 Power3s at University of Michigan (UMich)
- 32 160MHz IBM Azure Power2s at University of Texas (UTexas)
- 32 533MHz DEC Alpha EV56s at University of Virginia (UVa)
- 260 300MHz-nodes Cray T3E at SDSC
- 32 400MHz Sun HPC 10000s at SDSC

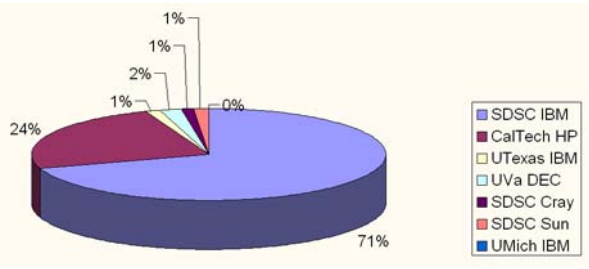

Figure 7. Breakup of CHARMM jobs completed

In the future, we intend adding the following resources:

- 88 300MHz-nodes Cray T3E at UTexas
- 32 400MHz dual-CPU Intel Pentium IIs at UVa

We estimate that if the user had used the resources available at his organisation alone (128 SGI Origins), it would have taken one month to complete what was complete in less than two days on the grid. The number of jobs run on each resource is shown in Figure 7. The vast majority of the jobs ran on the Blue Horizon at SDSC because that machine was by far the most powerful machine in the mix of available machines. Some of the machines did not contribute significantly to the results because of run-time problems.

Simplifying Grid Access. Legion's ease of use could be measured in what the user had to do as well as what he did *not* have to do to run his jobs. The user had to learn a mere four or five commands to run on the grid. The small number of commands is comparable to the number the user would have to learn for *each* queuing system had he not chosen Legion. During the experiment, the user did not have to log on to any of the queuing systems. He logged on to one machine at UVa on which Legion was installed. From a single shell on that machine, he initiated multiple jobs. Legion made the heterogeneous NPACI resources available to the user without his having to know the details of how to run on each resource. The heterogeneity of the resources extended in a number of dimensions:

- 6 organisations (UVa, TSRI, SDSC, UTexas, UMich, CalTech)
- 6 queue types (Maui, LoadLeveler, LSF, PBS, NQS)
- Up to 10 queuing systems
- Up to 6 architectures (IBM AIX, HP HPUX, Sun Solaris, DEC Linux, Intel Linux, Cray Unicos)

Identifying and Eliminating Problems. A number of run-time problems caused fewer total jobs to complete. Minor organisational problems aside, the problems we encountered fell into two categories: network slowdowns and site failures. The Legion run-time system suffered no problems during the experiment, although a number of potential extensions were identified. Also, although the CHARMM user used the grid heavily, the remaining users on the same grid were unaware of the experiment. While the experiment progressed, other Legion users continued to run their usual jobs on the grid.

Network Slowdown. During the experiment, we experienced slowdowns in the network connections between UVa and SDSC. From around noon through about 3PM US EST, transmission of medium-sized to large packets was difficult. Preliminary investigation showed that packets of size equal to or greater than 8800 bytes were lost entirely. Packets in the size range 8000-8800 bytes suffered over 90% loss rates. The loss rates for packets of size less than 8000 bytes were lower but still significant. The implication for Legion was that some messages between objects had to be retransmitted a number of times to ensure that they were received correctly. Consequently, for the CHARMM user, monitoring jobs became a slow process. At one point,

* Legion was not installed on the user's machines at The Scripps Research Institute (TSRI) because of site-specific firewall restrictions.

inquiring about the status of a job took nearly a minute to complete. Ordinarily, this process is almost instantaneous. Since the user could not monitor jobs quickly enough to start new ones, throughput was reduced.

Site Failures. Some of the NPACI sites experienced unforeseen failures. For example, at UMich, Legion encountered NFS failures. Since the ability to access permanent storage is important to Legion as well as CHARMM, the NFS failures reduced the throughput of CHARMM jobs. On the Blue Horizon machine at SDSC, the queuing system, Maui/LoadLeveler, had to be restarted a number of times because it became overloaded. During the time the queuing system was down, currently-running jobs continued to run. However, the queuing system could not inform anyone about the status about those jobs. Since "no information" is similar to what the queuing system reports when a job has been complete for a while, Legion assumed the jobs were complete and informed the CHARMM user accordingly. This erroneous reporting led the user to believe that it was safe to access the output files from the job. However, on analysis of these jobs, the user discovered that the output files were only partially complete. At UMich, the purge policy in place removed CHARMM files as well as persistent state required by Legion objects. Without their persistent state, Legion objects can behave erroneously. Likewise, without the appropriate input files CHARMM cannot run as intended.

5. Current Status of Parameter-Space Support in Legion

From the experiences of the CHARMM user and other p-space users we make some observations about grids and identify potential extensions to Legion. These extensions would enhance Legion's usability by building on low-level functionality already present. The first three of the following sub-sections present our observations; the last three present extensions we identified.

High-Level Services. The CHARMM experiment reassured us that a grid infrastructure must provide low-level functionality *and* high-level services. We consider it a significant advantage that using Legion, the user accessed heterogeneous resources controlled by multiple organisations with four or five commands and achieved order-of-magnitude speedup as compared to running at just one site. One example of a high-level service is the support for p-space studies. Without proper tools for initiating and monitoring large number of jobs, and collating the results of the jobs, users may find it difficult to run their p-space applications on a grid.

Human Factors. Three people were involved intimately with the continuous successful progress of the jobs: the user, a Legion liaison and an NPACI liaison. The Legion liaison was present in case problems arose with Legion itself during the execution. Since Legion itself suffered no run-time problems, this person used Legion tools to identify site-specific problems as they arose. The NPACI liaison coordinated on-site efforts to keep the experiment running. Finally, administrators at individual sites ensured that problems were resolved as soon as possible by correcting misconfigurations, restarting services, increasing quotas, etc. Although this collaboration was rewarding, in the future the involvement of all parties except the user must be eliminated.

Site Services. The number of site failures that were identified was astonishingly high. Normally, users never expect services such as queues and operating systems to fail. Likewise, users rarely consider network failures when running their applications. However, running large numbers of high-performance jobs can stress-test every component of a grid. We discovered previously-ignored limits on the number of jobs queues can manage, queue-imposed job duration limits, credential expirations with file systems, purge policies, process table limits, quota exhaustions and numerous other problems, each of which could make a site unusable for continued running.

Graceful Error Handling. Legion has been designed to mask many kinds of failures from end-users. While this strategy usually benefits the user, sometimes it is important for the grid infrastructure *not* to mask failures from the user. For example, the network failures discussed earlier were masked from the user who saw only gracefully-degraded performance. However, Legion also masked most site failures from the user, which often conveyed the mistaken impression that Legion itself had failed. Consequently, we are reviewing all aspects of error handling and propagation in Legion.

Support for Archiving. Although Legion permits users to specify input and output files at any time during the execution of a job, archival support is almost non-existent. In particular, there is no way for a user to specify that some files are meant to be stored on some kind of long-term storage after the job is complete. Instead, the Legion file solutions are that after a job is complete, the files are either copied out to the user's local directories, or to Legion's own distributed shared file system, or deleted. None of these solutions is satisfactory for jobs that generate large amounts of data. The user's local directories or the individual components of the distributed file system may not have space to store large amounts of data. Moreover, the user may not want to copy files out the moment the job is done. Instead, scientific users generating large amounts of data, such as the CHARMM user, are likely to want to archive the data generated by their jobs on some long-term storage and access the data at their leisure. Since Legion developers did not anticipate

such a need, currently, archiving has to be done by users themselves as part of their jobs.

Web Interfaces. We have developed a web portal that scientists can use to run jobs such as CHARMM. The Legion Grid Portal is an interface to a grid system. Users interact with the portal, and hence a grid through an intuitive interface from which they can view files, submit and monitor jobs, and view accounting information. The architecture of the portal is designed to accommodate multiple diverse grid infrastructures, legacy systems and application-specific interfaces. The current implementation of the Legion Grid Portal is with familiar web technologies over the Legion grid infrastructure. The portal can be extended in a number of directions — additional support for grid administrators, greater number of application-specific interfaces, interoperability between grid infrastructures, and interfaces for programming support. The portal has been in operation since February 2000 on *npacinet*, our worldwide grid managed by Legion on NPACI resources [23].

6. Conclusion

The success of a grid system depends on how easily and securely it permits users to perform their computations by collaborating and accessing available resources. A key component of a grid system is software that presents users with abstractions of resources. Legion provides those abstractions *via* uniform, easy-to-use interfaces. These interfaces, ranging from tool-level to programming-level, greatly reduce the difficulties of computing in distributed, heterogeneous environments. The mechanisms underlying the interfaces enable users to perform cross-machine, cross-architecture and cross-organisation computation. By enabling such computations on a large scale, Legion supports capacity computing. Legion's flexible and extensible object model supports capability computing by permitting novel methods of computation.

Legion is a suitable environment for running large numbers of high-performance jobs on a grid, as demonstrated by the CHARMM experiment. Legion provides a suite of tools for a grid that are similar to what traditional operating systems provide for a single system. Using these tools, users can start, monitor and terminate jobs on remote machines in a straightforward manner. Legion masks unwanted detail from the user, thus permitting him to focus on completing his work.

Legion consists of 350,000 lines of code and has been ported to Windows NT as well as a large number of Unix variants, including Linux (Intel, Alpha), Unicos (T90, T3E), AIX (SP-2, SP-3), HPUX, FreeBSD, IRIX (Origin 2000) and Solaris (Enterprise 10000). Legion has been integrated with a large number of queuing systems, such as PBS, LSF, Codine, LoadLeveler and NQS. It has been deployed on machines belonging to NSF-PACI, NASA IPG and the DoD MSRCs. Currently, Legion is running at over 300 hosts across the United States and Europe. Researchers using Legion currently are from a number of disciplines, such as biochemistry (e.g., complib, a protein and DNA sequence comparison), molecular biology (e.g., CHARMM, a p-space study of 3D structures), materials science (e.g., DSMC, a Monte Carlo particle-in-cell study), climate modelling (e.g., BT-MED, a 2D barotropic ocean model), aerospace (e.g., flapper, a p-space study of a vehicle with flapping wings), astronomy (e.g., Hydro, a study of a rotating gas disk around a black hole), neuroscience (e.g., a biological-scale simulation of a mammalian neural net), information retrieval (e.g., PIE, a personalised search environment) and computer graphics (e.g., a p-space rendering of independent movie frames).

We expect users to become more accustomed to using distributed resources, often in ways not anticipated today. Legion's architecture promises to satisfy grid demands of the present as well as the future.

7. References

[1] —, *The Legion Manuals (v1.7)*, Univ. of Virginia, Oct. 2000.

[2] Bayucan, A., Henderson, R. L., Lesiak, C., Mann, N., Proett, T., Tweten, D., *Portable Batch System: External Reference Specification*, Tech. Rep., MRJ Technology Solutions, Nov. 1999.

[3] Brooks, B. R., Bruccoleri, R. E., Olafson, B. D., States, D. J., Swaminathan, S., Karplus, M., *CHARMM: A Program for Macromolecular Energy, Minimization, and Dynamics Calculations*, J. Comp. Chem., Vol. 4, 1983.

[4] Chapin, S. J., Katramatos, D., Karpovich, J. F., Grimshaw, A. S., *Resource Management in Legion*, Tech. Rep. CS-98-09, Univ. of Virginia, Feb. 1998.

[5] Ferrari, A. J., Grimshaw, A. S., *Basic Fortran Support in Legion*, Tech. Rep. CS-98-11, Univ. of Virginia, Mar. 1998.

[6] Ferrari, A. J., Knabe, F., Humphrey, M. A., Chapin, S. J., Grimshaw, A. S., *A Flexible Security System for Metacomputing Environments*, High Perf. Computing and Networking Europe, Apr. 1999.

[7] Foster, I., Kesselman, C., *The Grid: Blueprint for a New Computing Infrastructure*, Morgan Kaufmann, 1999.

[8] Geist, A., Beguelin, A., Dongarra, J., Jiang, W., Manchek, R., Sunderam, V., *PVM: Parallel Virtual Machine: A User's Guide and Tutorial for Networked Parallel Computing*, MIT Press, 1998.

[9] Grimshaw, A. S., Ferrari, A. J., West, E., *Mentat*, Parallel Programming Using C++, MIT Press, 1996.

[10] Grimshaw, A. S., Wulf, W. A., *The Legion Vision of a Worldwide Virtual Computer*, Comm. of the ACM, Vol. 40, No. 1, Jan. 1997.

[11] Grimshaw, A. S., Lewis, M. J., Ferrari, A. J., Karpovich, J. F., *Architectural Support for Extensibility and Autonomy in Wide-Area Distributed Object Systems*, Tech. Rep. CS-98-12, Univ. of Virginia, Jun. 1998.

[12] Grimshaw, A. S., Ferrari, A. J., Lindahl, G., Holcomb, K., *Metasystems*, Comm. of the ACM, Vol. 41, No. 11, Nov. 1998.

[13] Grimshaw, A. S., Ferrari, A. J., Knabe, F., Humphrey, M. A., *Wide-Area Computing: Resource Sharing on a Large Scale*, IEEE Computer, Vol. 32, No. 5, May 1999.

[14] Gropp, W., Lusk, E., Doss, N., Skjellum, A., *A High-Performance, Portable Implementation of the Message Passing Interface Standard*, Par. Computing, Vol. 22, No. 6, Sep. 1996.

[15] Hempel, R., Walker, D. W., *The Emergence of the MPI Message Passing Standard for Parallel Computing*, Comp. Stds. and Interfaces, Vol. 7, 1999.

[16] Howard, J., Kazar, M., Menees, S., Nichols, D., Satyanarayanan, M., Sidebotham, R., West, M., *Scale and Performance in a Distributed File System*, ACM Trans. on Computer Syst., Vol. 6, No. 1, Feb. 1988.

[17] International Business Machines Corporation, *IBM LoadLeveler: User's Guide*, Sep. 1993.

[18] Karpovich, J. F., Grimshaw, A. S., French, J. C., *Extensible File Systems (ELFS): An Object-Oriented Approach to High Performance File I/O*, 9th Annual Conf. on Object-Oriented Programming Syst., Lang. and App. (OOPSLA), Oct. 1994.

[19] Kingsbury, B. A., *The Network Queueing System (NQS)*, Tech. Rep., Sterling Software, 1992.

[20] MacKerell, A. D.. Jr., Brooks, B. R., Brooks, C. L. III, Nilsson, L., Roux, B., Won, Y., Karplus, M., *CHARMM: The Energy Function and Its Parameterization with an Overview of the Program*, The Encycl. of Comp. Chem., Vol. 1, 1998.

[21] Natrajan, A., Humphrey, M. A., Grimshaw, A. S., *Capacity and Capability Computing in Legion*, The 2001 Intl. Conf. on Computational Sc., May 2001.

[22] Natrajan, A., Crowley, M., Wilkins-Diehr, N., Humphrey, M. A., Fox, A. D., Grimshaw, A. S., Brooks, C. L. III, *Studying Protein Folding on the Grid: Experiences using CHARMM on NPACI Resources under Legion*, 10th Intl. Symp. on High Perf. Dist. Computing, Aug. 2001.

[23] Natrajan, A., Nguyen-Tuong, A., Humphrey, M. A., Grimshaw, A. S., *The Legion Grid Portal*, Grid Computing Environments 2001, Concurrency and Computation: Practice and Experience, 2001.

[24] Nguyen-Tuong, A., *Integrating Fault-tolerance Techniques in Grid Applications*, Ph.D. Diss. CS-2000-05, Univ. of Virginia, Aug. 2000.

[25] Sandberg, R., Goldberg, D., Kleiman, S., Walsh, D., Lyon, B., *Design and Implementation of the SUN Network File System*, Proc. of USENIX Conf., 1985.

[26] Seigel, J., *CORBA Fundamentals and Programming*, Wiley, ISBN: 0471-12148-7, 1996.

[27] Snir, M., Otto, S., Huss-Lederman, S., Walker, D., Dongarra, J., *MPI: The Complete Reference*, MIT Press, 1998.

[28] Weissman, J., *Scheduling Parallel Computations in a Heterogeneous Environment*, Ph.D. Diss. CS-1995-06, Univ. of Virginia, Aug. 1995.

[29] White, B. S., Grimshaw, A. S., Nguyen-Tuong, A., *Grid-Based File Access: The Legion I/O Model*, High Perf. Dist. Computing 9, Aug. 2000.

[30] Zhou, S., *LSF: Load Sharing in Large-scale Heterogeneous Distributed Systems*, Proc. of Work. on Cluster Computing, Dec. 1992.

[31] Zhou, S., Wang, J., Zheng, X., Delisle, P., *Utopia: A Load Sharing Facility for Large, Heterogeneous Distributed Computer Systems*, Soft. Prac. and Exp., Vol. 23, No. 2, 1993.